Integrazione di Sistemi Embedded
Laboratorio 04

Matteo Perotti 251453
Giuseppe Puletto 251437
Luca Romani 255244
Giuseppe Sarda 255648

November 25, 2018

1 Introduzione

Il quarto laboratorio ha come obiettivo l'esplorazione delle funzionalità che il comando **make** può offrire tra le quali il testing e la generazione automatizzata del file eseguibile. Strumento fondamentale per questa fase è il file **Makefile** usato da make.

2 Approccio alla scrittura del Makefile

Per la scrittura del Makefile, il quale contiene tutte le "ricette" necessarie per la completa compilazione di un progetto, si è consultato il manuale GNU relativo al comando *make* reperibile alla pagina web GNU make.
Di seguito sono riportate le informazioni più rilevanti ricavate dal manuale, alcune delle quali sono state usate all'interno dei Makefile prodotti.

Phony targets

Usando il "token" **.PHONY** si riesce ad indicare un target che non corrisponde al nome di un vero e proprio file, ma ad una ricetta eseguita soltanto se espressamente richiesto. Questa direttiva è utile in caso di omonimia tra un target e un file, i quali però non sono logicamente collegati, e che quindi, senza un'indicazione specifica, sarebbero erroneamente vincolati. Dichiarando dunque:

```
.PHONY: clean
clean:
        #do something
```

il target *clean* viene eseguito esclusivamente quando richiesto all'esecuzione del comando make evitando eventuali malfunzionamenti nel caso esita un file nominato "clean" nella directory del progetto.

Versioni del Makefile

Il Makefile è stato sviluppato in diversi modi, utilizzando regole esplicite ed implicite e variando il compilatore e le opzioni di compilazione in modo statico oppure dando la possibilità di scelta all'utente.

Versione 1 In questa versione del makefile è stata definita una variabile "CC" per rendere più leggibile il file che di default imposta come compilatore gcc.
Il make file ha un target fittizio "all" senza ricette e con un'unica dipendenza: il file eseguibile che si vuole creare, universalDrawer in questo caso. In questo modo il makefile cerca subito quali sono i target che servono per la dipendenza dell'eseguibile.
Volendo strutturare il file in modo esplicito, è stato definito un target per ogni file oggetto utile a generare l'eseguibile. Tra le dipendenze di ogni file oggetto sono stati inseriti i file .c e .h necessari per la compilazione. Come ultimo target è stato definito "clean" che è senza dipendenze e che semplicemente rimuove, se richiesto, tutti i file oggetto e il file eseguibile presenti nella stessa directory del makefile

Versione 2 Relativamente alle versione 2 non ci sono sostanziali differenze rispetto alla precedente versione. Unica variazione si ha nell'architettura target di compilazione che, come da specifiche, deve essere per ARM.

Versione 3 Il terzo makefile assolve le specifiche richieste mediante l'uso di variabili e compilazione condizionale. Dapprima si controlla se la variabile PREFIX è stata definita all'esecuzione del comando make: se sì, viene utlizzato come compilatore $(PREFIX)-gcc, in caso contrario gcc. La definizione o meno di PREFIX condiziona il valore della variabile OPZIONE. Se si definisce PREFIX=arm-none-eabi, viene eseguito il compilatore arm-none-eabi con la specifica addizionale –specs=nosys.specs. Tutto questo è possibile mediante l'uso di un'ulteriore compilazione condizionale che fa uso delle variabili COMPILATORE_1 e COMPILATORE_2. Qualunque sia il compilatore scelto, la compilazione viene eseguita sempre con l'opzione -static. Il Makefile si serve anche di altre quattro variabili, al fine di specificare il percorso dei file.c, quello dei file.h, e le

cartelle di destinazione dei file oggetto ed eseguibile. Quest'ultime vengono create rispettivamente attraverso la ricetta del target main.o e del target universalDrawer. Il target clean rimuove ricorsivamente le cartelle creata e i file oggetto ed eseguibile al loro interno. Anche in questo Makefile si fa uso del target fittizio all.

Versione 4 Il Makefile è stato fatto per sfruttare il più possibile le regole implicite. Tutti i files .c vengono tradotti nei corrispettivi files oggetto .o. Quando viene chiamato il compilatore vengono aggiunte delle opzioni specifiche differenti da quando viene chiamato il linker (tutto racchiuso in variabili). Come nella precedente versione è possibile impostare la toolchain da utilizzare esplicitando il valore della variabile PREFIX direttamente quando viene chiamato il comando make. E' stato previsto anche un target phony "test" (per ora lasciato vuoto) per poter rendere agevole il test del programma eseguibile, ma la ricetta per questo target verrà scritta nel Makefile successivo. Come per le versioni precedenti anche questo Makefile contiene un phony target "clean" per gestire la rimozione dei files oggetto ed eseguibili, ad esempio qualora si volesse modificare la toolchain da una volta all'altra. In ogni ricetta relativa a target non-phony è presente anche un comando mkdir per creare la folder relativa: l'opzione -p evita che venga dato errore qualora la cartella fosse già presente.

3 Implementazione del test

Per la realizzazione del target test si è deciso di dividere la verifica del codice scritto per il precedente laboratorio in 4 parti:

- Creazione di uno script Python in grado di generare comandi di disegno casuali sotto specifiche
- Scrittura di un codice Python, basato sugli identici algoritmi usati durante il precedente lab, che si comportasse secondo specifiche relative alla parte funzionale ma che lasciasse totale libertà implementativa
- Esecuzione dei comandi generati al punto uno da parte del codice C e dello script Python
- Confronto dell'output finale delle due esecuzioni

Si noti che i limiti di questo test stanno nel fatto che, nel caso in cui gli algoritmi di disegno fossero sbagliati in partenza, non c'è modo di accorgersene. I casi limite inoltre non vengono così testati perché i comandi al punto 1 vengono per definizione prodotti al fine di rispettare le specifiche.

Il codice C viene testato generando 2000 comandi casuali.

Modifica del programma

Il programma è stato modificato per poter riconoscere il comando di QUITTING ('Q'). Appena esso viene ricevuto viene chiamata una funzione che apre un file, disegna la matrice (frameBuffer) e poi lo chiude. In questo modo diventa semplice il confronto tra le due matrici generate dai programmi scritti in diversi linguaggi. L'interfaccia del programma è stata anche regolarizzata per rientrare nelle specifiche di interfaccia.

Generazione dei comandi

Lo script per la generazione dei comandi accetta dall'utente un certo numero di comandi che vengono creati usando la funzione *randint* contenuta nel modulo *random*. Infine i comandi generati vengono scritti all'interno di un file che verrà poi passato ai file eseguibile. L'esecuzione finisce con la scrittura di un **token** (Q) che segnala la fine dello stream di comandi. All'interno del .txt che viene poi gestito dall'eseguibile del progetto C, lo script Python inserisce, tra i comandi che sono di default corretti, del "rumore casuale" rappresentato da caratteri ASCII di qualsiasi tipo: lettere, numeri, punteggiatura o *white spaces*. Si noti che in questo insieme non è presente il comando di uscita Q. Di seguito si riporta il codice python per una più immediata ed esemplficativa comprensione accompagnato da un esempio di file di comandi.

```
#MAX_X e' un numero massimo di caratteri generabili

def noiseGenerator(out):
        time=random.randint(0,MAX_X)
        i=0
        while i < time:
                char=random.choice(string.printable)
                if char != 'Q':
                        out.write(char)
                i+=1
#a fine generazione di un comando con probabilita' del 33%
#viene chiamata la funzione noiseGenerator

noise=random.randint(0,2)
if noise == 0:
        noiseGenerator(f)


#esempio di file di comandi

P1130150
L0090580310551
D`+V,xm|(hUohlx#utO7gv^=Tk^"W#DB2dG<0SshG5Z-gakrG>>EkMA
O|n1,UYt=YKn[N \` Zl3t`Mf+uymAg-i.<O6yo1pi[s:304].
WqZPx-RTrbBLe)5hL1150370911210
P0400540
Sgc_lFHO            hywq?V_C^r87Y/u>M&th031h
=DN,)(roCBrL0PbS0.#Py     JqE0941031060490
Q
```

Automazione del test con Makefile

Il makefile è stato aggiornato con nuove dipendenze per processare i files aid.c e aid.h (e shared.h che ora viene condiviso anche con main.c). Il test è stato integrato nel Makefile utilizzando un altro phony target di nome "test". Esso contiene come ricetta delle istruzioni bash che eseguono lo script python per la generazione dei comandi casuali, poi lo script python per la generazione del file di comparazione (che prende gli ingressi dal file appena generato). Il file con i comandi viene reindirizzato anche come stdin per il programma main da testare, in questo modo con una semplice diff è possibile vedere a terminale se sono presenti incongruenze.